Max Leuthäuser

Pure Embedding of Evolving Objects

Erstveröffentlichung in / First published in:

ADAPTIVE 2017: The Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications, Athens, 2017. ThinkMind Digital Library, S. 22 – 30. ISBN 978-1-61208-532-6

Pure Embedding of Evolving Objects

Max Leuthäuser

Software Technology Group
TU Dresden
Email: max.leuthaeuser@tu-dresden.de

***Abstract*—Scripting languages are extraordinarily popular due to their very flexible object model. Dynamic extensions (i.e., adding, removing and manipulating behavior and state) allow for the evolution and adaption of objects to context changes at runtime. Introducing this flexibility into a statically typed, object-oriented language would improve programmability and separation of concerns beyond the level of what one could usually gain with inheritance, mixins, traits or manually adapted design-patterns. They often lead to object-schizophrenia or the need for hand-crafted, additional management code. Although there were already attempts bringing flexible objects into statically typed languages with the benefits of an explicitly crafted core calculus or type system, they need their own compiler and tooling which limits the usability, e.g., when dealing with existing legacy code. This work presents an embedding of dynamically evolving objects via a lightweight library approach, which is *pure* in the sense, that there is no need for a specific compiler or tooling. It is written in Scala, which is both a modern object-oriented and functional programming language. Our approach is promising to solve practical problems arising in the area of dynamical extensibility and adaption like role-based programming.**

Keywords–Scala; evolving objects; object-oriented programming; dispatch.

I. INTRODUCTION

Scripting languages like Python, JavaScript, Ruby, Perl or Lua offer very flexible object semantics to the developer. On the one hand side, programmers can rely on classical object-oriented features, such as inheritance, encapsulation and polymorphism, and on the other, they are able to add and remove members (e.g., attributes and functions) from existing objects or merge them at any given point in their life-cycle [1].

This is usually not available in statically typed object-oriented languages. Imagine you have a client that wants to execute some behavior at a (core-) object of interest but that desired behavior is not available (Fig. 1). Using inheritance, mixins, traits or design-patterns is not desirable. The first three techniques will result in a very static system design and exponentially many classes, while the use of patterns often leads to object-schizophrenia [2] and the need of additional management code. Adding and removing members from existing objects at runtime are indeed very useful operations for todays software-systems, that have a very high demand for adaptivity and need to cope with complexity and change [3].

Is bridging the gap between statically-typed, object-oriented languages and evolving objects at runtime via pure embedding possible without too much effort? To answer that, the main contributions of this paper are:

- An introduction and summarizing technological overview on *SCROLL* [4], a lightweight library that allows for pure embedding of evolving objects in a modern, statically typed object-oriented language (Scala [5]), utilizing only those features that are available through its standard compiler. This library itself is small (∼1400 lines of code), allows for easy integration of legacy code and a high separation of concerns. It is limited on the side of type-safety as one might expect. Nevertheless, having a statically-typed host language for evolving objects supports the developer with the best of both worlds: static typing leads to an earlier detection of programming mistakes through static code analysis, better documentation in form of type-signatures, compiler-optimization, runtime-efficiency and an improved design-time development experience, while the latter supports easy prototyping, change to unknown requirements or unpredictable data and application integration. In summary: "*Static typing where possible, dynamic typing when needed!*" [6].
- An abstraction of that library into a more general implementation pattern by requiring only three fairly basic techniques to the host language.
- An example application showing that dynamically evolving objects are useful in the domain of role-based programming.

Scala was chosen as host language for *SCROLL* not only because of its combination of object-oriented and functional programming features, but as well due to its scalability and interoperability with the Java virtual machine providing easy integration of legacy code and the use of already established tools. *SCROLL* in particular takes advantage of Scala's features such as higher order functions, general operator notations, flexible syntax, implicits, compiler rewrites and implicit definitions of parameters.

The remainder of the paper is structured as follows. First, we will introduce in Sec. II the way evolving objects can be implemented with *SCROLL*. Additionally, the most important application programming interface- (API-) calls are explained. Following that, the actual implementation is described and will be abstracted into a more general implementation pattern by laying out its required three basic techniques (Sec. III). The abstraction from roles to evolving objects is demonstrated in Sec. IV and shows how role-based programming can be handled as well. Finally, the *SCROLL* approach is compared to more naive solutions using various design-patterns (Sec. V) and other coeval approaches from the related work (Sec. VI).

 ISBN: 978-1-61208-532-6

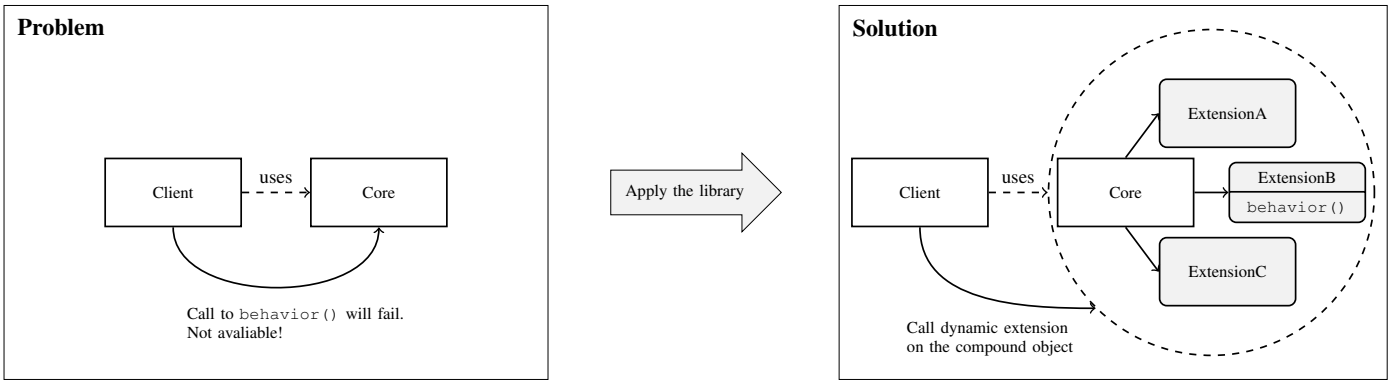


Figure 1. *Problem*: Imagine you have a client that wants to execute some behavior at a (core-) object of interest but that desired behavior is not available (see *left box*). *Solution*: Applying the library allows for dynamically adding new behavior at runtime while wrapping all the extension parts (`ExtensionA`, `ExtensionB` and `ExtensionC`) of the augmented object (`Core`) into one logical *compound object* (see *right box*).

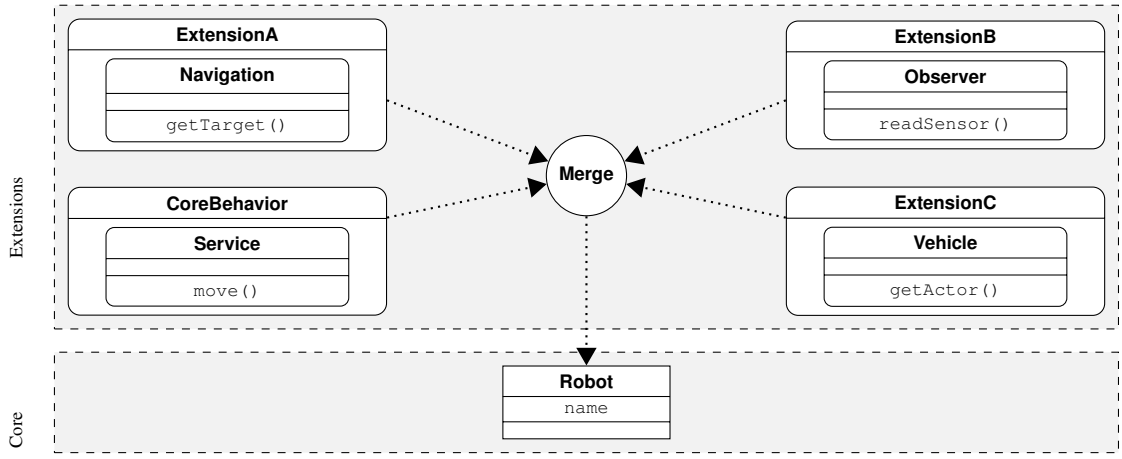


Figure 2. The class `Robot` is constructed (dotted arrows) from different extensions and acquires the contained behavior.

II. Evolving Objects in *SCROLL*

This section provides a brief introduction into the way one can use evolving objects as provided by *SCROLL* by example (see Fig. 2 and 3). A standard Scala case class (`Robot`) should be augmented with new behavior encapsulated in three different extensions (`ExtensionA`, `ExtensionB` and `ExtensionC`). Each of them provides a new aspect of the robot via functions (like finding a target to move to or observing sensor values) attached to case classes. This allows for a high degree of separation of concerns with multiple hierarchically nested components. The core behavior (with case class `Service`) aggregates all of the provided functionality without having to worry about its actual location. There are several calls to *SCROLL* in the example. Those shall be explained in the following:

- `+-Operator` (e.g., at line 6 in Fig. 3): In Scala, method calls can be written as infix operators. `+this` is equivalent to `this.+()`. Because extensions should be merged into any given object, we cannot assume that this object actually provides this +-operator. Thus, Scala's implicit conversion [7] is used to wrap the core object into an equivalent *compound object* exposing the required programming interface. In summary, by calling the +-operator the developer is able to forward arbitrary calls to some extension he assumes should be available on the core object without worrying about their actual location. The function-lookup resolution technique is explained in more depth in Sec. III.
- `play` (e.g., at line 32 in Fig. 3): This method attaches the selected behavior to the core object. The name stems from role-based programming, where roles can be seen as some kind of dynamic extension (Sec. IV). There, playing a role is equivalent to acquiring its behavior and state.
- `Compartment` (e.g., at line 3 in Fig. 3): A compartment is an objectified collaboration with a limited number of participating roles and a fixed scope [8] and stems from role-based programming as well. It was introduced to clearly distinguish from the heavily overloaded term *context*. While a context (e.g., a cold and rainy day in London) is prescriptive, without its own identity, intrinsic behavior or existential parts and with an indefinite lifetime - a compartment (e.g., a first-class train car) is descriptive. Its instances carry identity, have behavior, state, a defined lifetime and contain roles as its parts. Mixing in the `Compartment` trait exposes *SCROLL*'s basic programming interface to the current class. Contained classes or case classes can be seen as containers for new behavior and state that should be attached later on. From the developer's point of view, one could rewrite the introductory sentence to "a compartment is an objectified collaboration with dynamic behavior and state and a fixed scope".

 ISBN: 978-1-61208-532-6

- `merge` (e.g., at line 33 in Fig. 3): The relationship between a core object and all of its extensions is stored compartment-specific (as explained in Sec. III-C). So, executing behavior spanning over multiple extensions like in `CoreBehavior` requires a merging of all the participating extensions (i.e. compartments) with their specific storage.

When running the example code, the console output as shown in Fig. 3 will be generated. There are several slightly more advanced examples available online [9].

```
1  case class Robot(name: String)
2
3  object CoreBehavior extends Compartment {
4    case class Service() {
5      def move() {
6        val name: String = +this name()
7        val target: String = +this getTarget()
8        val sensorValue: Int = +this readSensor()
9        val actor: String = +this getActor()
10       info(s"I am $name and moving to the $target
             ↪ with my $actor w.r.t. sensor value of
             ↪ $sensorValue.")
11     }
12   }
13 }
14
15 object ExtensionA extends Compartment {
16   case class Navigation() {
17     def getTarget = "kitchen"
18   }
19 }
20
21 object ExtensionB extends Compartment {
22   case class Observer() {
23     def readSensor = 100
24   }
25 }
26
27 object ExtensionC extends Compartment {
28   case class Vehicle() {
29     def getActor = "wheels"
30   }
31 }

32 val myRobot = Robot("Pete") play Service() play
       ↪ Navigation() play Observer() play
       ↪ Vehicle()
33 CoreBehavior merge ExtensionA merge ExtensionB
       ↪ merge ExtensionC
34 myRobot move()

35 I am Pete and moving to the kitchen with my
       ↪ wheels w.r.t. sensor value of 100.
```

Figure 3. The robot is constructed from multiple extensions dynamically at runtime. Running the example generates the console output shown above.

III. Implementation

This section explains the basic technologies used by *SCROLL* for the pure embedding of evolving objects. Together, these form an implementation pattern that is useful for adapting this library approach to other host languages.

A. Implicit Conversions

We want to be able to mix in extensions to any given object of any type in Scala. Implicit conversions [7] provide a lightweight way to expose *SCROLL*'s programming interface for adding, removing and transferring behavior or state to any object. Listing 1 gives a brief excerpt.

```
1  implicit class Player[T](val wrapped: T) {
2    /* Applies lifting to Player */
3    def unary_+ : Player[T] = this
4
5    def play(role: Any): Player[T] = /* ... */
6    def drop(role: Any): Player[T] = /* ... */
7    def transfer(role: Any) = new {
8      def to(player: Any) { /* ... */ }
9    }
10
11   /* ... */
12   override def equals(o: Any) = /* ... */
13 }
```

Listing 1. The generic `implicit` class `Player`.

Scala's implicit conversion is used to wrap the core object into an equivalent *compound object* exposing the required API in a type-safe manner. Furthermore, the issue of *object-schizophrenia* needs to be addressed with a clear notion of object identity. This term has not been introduced explicitly by any publication, but appeared in a set of web-pages in the field of context-oriented programming and can be described like this: "*Object Schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity).*" [10]. It can be seen as another instance of the *split object problem* [11]. Here, the identity of an object should be the same independent of which extension is attached. Consequently, object identity should reflect this properly. Four kinds of comparison are possible:

1) `core == core + extension`
2) `core + extension == core`
3) `core + extension == core + extension`
4) `core + extensionA == core + extensionB`

To implement this, we modify the identity-related method of the compound object represented by `Player` as shown in the above code-listing. In fact, `==` and the `equals`-method are equivalent in Scala. That is, the expressions `x == y` and `x.equals(y)` give the same result. We define the `equals`-method in such way that it maps to the implementation of the core object, and, in case the right-hand operator of `==` is an evolving object as well, compare with its core object. This solves the problem for expressions 2 to 4, but unfortunately does not for expression 1, since we cannot modify the `equals`-method of arbitrary objects using a library approach. If the comparison of a plain core object is required apply the `+-Operator` (see Sec. II) to it. This will trigger the dynamic conversion using the implicit class `Player` and applies the desired comparison.

B. Dynamic Trait

Behavior and state from extensions that is not natively available to the core object needs to be addressed somehow. Scala's `Dynamic` trait [12] is used to implement that behavior. Once the proper extension is identified and selected (see Sec. III-C and III-D) the actual invocation should take place. To do so, calls to extension-specific functionality, that

would normally fail during type-checking phase, are rewritten according to the rules by the compiler itself [13] as shown in Listing 2:

```
foo.method("blah")
  ⇝ foo.applyDynamic("method")("blah")

foo.method(x = "blah")
  ⇝ foo.applyDynamicNamed("method")(("x",
      ↪ "blah"))
foo.method(x = 1, 2)
  ⇝ foo.applyDynamicNamed("method")(("x", 1),
      ↪ ("", 2))

foo.field
  ⇝ foo.selectDynamic("field")
foo.variable = 10
  ⇝ foo.updateDynamic("variable")(10)
```

Listing 2. Compiler rewrite rules from the `Dynamic` trait [13].

That is exactly the point where type safety is lost. The actual set of dynamic extensions that are bound to the core object is not statically known, hence static type-safety is not available. As an example, the method call to the robots `name` attribute from Fig. 3 (line 6) is translated as shown in Listing 3.

```
+this.name
  ⇝ this.unary_+().name
  ⇝ new Player[Robot](this).name
  ⇝ new Player[Robot](this).selectDynamic("name")
```

Listing 3. Rewriting for dynamical access to the `Robot` attribute `name`.

SCROLL hooks into those rewritten methods and triggers the actual invocation and error handling. We refrain from using runtime exceptions or similar exception-based error handling in case of not being able to find the functionality the developer is querying for. Instead, Scala's `Either` container type is used by the library. It has two sub types, `Left` and `Right`. If an `Either[A,B]` object contains an instance of `A`, then the `Either` is a `Left`. Otherwise, it contains an instance of `B` and it is a `Right`. Although, there is nothing in the semantics of this type that would specify one or the other sub type to represent an error or a success, by convention it is used to carry the error case as `Left` (e.g., `DynamicBehaviorNotFound`), whereas the `Right` contains the success value (i.e., the result of executing the dynamic behavior). Together with a sealed type hierarchy with data types using case classes that represent errors, very readable messages compared to actual stack-traces from standard Java exceptions are generated.

C. Graph-based Backend

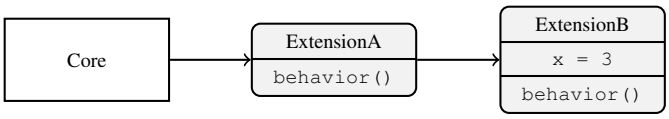
In *SCROLL*, a graph is used for storing the relations between core objects and its extension instances. That allows for easy querying of extension-specific behavior that was attached to the core object. Furthermore, there are many graph libraries implementing various aspects like caching and distribution. In general, a (labeled) graph H is a 4-tuple (V, E, Lab, L_Σ) with:

- V is a finite set of vertices (nodes) with $|V| \geq 0$,
- E is the set of edges, where E is a relation $E \subseteq V \times V$,
- L_Σ is the set of labels and
- $Lab : V \cup E \rightarrow L_\Sigma$ is the labeling function, which assigns a label to each node in V and edge in E.

For *SCROLL* this can be adapted to:

- V is the set of objects (core and all extension instances),
- E is the set of relations between core objects and its extension instances,
- L_Σ is the set of type names for all objects in V and
- $Lab : V \rightarrow L_\Sigma$ assigns each object in V its type.

JGraphT [14] was chosen as underlying graph library providing already the necessary graph-theory objects like predefined edge- and node-types, as well as simple algorithms for traversing the graph. *SCROLL* abstracts from that making it easy to plug-in any other convenient library, e.g., for easy scaling or distribution of the graph if required. Another implementation making use of Google's Guava framework [15] for caching is available, too. That speeds up querying the core for the actual behavior hidden in some extension if asked by the client. Additionally, with Kiama's pure embedding of attribute-grammars in Scala [16], a third backend is provided. Querying and updating the graph is implemented as (cached-) attributes and rewrites respectively.




```
implicit val dispatchDescription =
  From(_.isInstanceOf[Core]).
  To(anything).
  Through(anything).
  Bypassing(_ match {
    case ExtensionA() => true
    case ExtensionB(x) if x == 3 => false
    case _ => true // default case
  })
```

Figure 4. An example for the need of customizable dynamic dispatch.

D. Customizable Dynamic Dispatch

Dispatching in adaptive systems is context-dependent. Selecting the appropriate extension that should be selected for answering a call to the required behavior may be ambiguous. The developer should be able to key out the desired selection. *SCROLL* supports this with function composition and Scala's pattern matching making use of an explicit dispatch description which is passed down to the actual method invocation as implicit argument. The given selection functions are applied while traversing the graph-based storage holding the relations between core objects and its extension instances. See Fig. 4 for an example. We construct a new dispatch description using four factory methods provided by the API and pass our selection functions into them. We are only interested in our core object (`_.isInstanceOf[Core]`) so we are using this for the `From`-selector. Now, lets assume we do not care for the types of extensions that are actually around so we pass `anything` to the `To`-selector, which will always evaluate to `true`, so every extension instance will be considered while traversing the graph. Same goes for `Through` on intermediate nodes. Finally, for the `Bypassing`-selector we want to define that an instance of `ExtensionB` with the state `x = 3` should be selected, hence never bypassed. With an explicit

 ISBN: 978-1-61208-532-6

dispatch description, the developer defines a sub-graph of the underlying graph as follows: let $H = (V, E, Lab, L_\Sigma)$ and $H' = (V', E', Lab', {L_\Sigma}')$ be those storage graphs. Then H' is a sub-graph generated for dispatching out of H with $H' \subseteq H$, if $V' \subseteq V$, $E' \subseteq E$, $Lab' \subseteq Lab : V' \cup E' \rightarrow {L_\Sigma}'$ and ${L_\Sigma}' \subseteq L_\Sigma$.

E. The SCROLL Implementation Pattern

So far, we have shown how arbitrary objects can be augmented dynamically with new functionality or state grouped together in extension. Moreover, obstacles arising from object-schizophrenia can be solved with a compound object enabled by dynamic conversions and an adapted notion of object identity (i.e., the identity of an object should be the same independent of which extension is attached). Using Scala's `Dynamic` trait together with a graph-based backend allows for easy querying for behavior that is not natively available at the core object. How to transfer and adapt this? An implementation pattern is a reusable and adaptable solution to a certain problem and shows best practice for developers while offering insights to relationships and interactions between its components.

As introduced in the previous subsections, *SCROLL* requires the concept of a *marker trait* (1) for triggering compiler rewrites handing over calls to the library for finding behavior that is not natively available at the core object. For assembling a compound object from the core plus all its extension *implicit conversions* (2) are needed. The storage of the relationships between each individual core object and its extensions can be done easily with any *graph-based backend* (3), or alternatively with tables or maps. If one is able to find or emulate these three techniques in the desired statically-typed, object-oriented language it is easy to provide an alternative implementation of *SCROLL*.

IV. Separation of Concerns

The main goal of having extensions to dynamically evolve objects is the separation of different entity concerns while being able to plug them altogether at runtime if a context-dependent change to the system occurs. This not only increases adaptability of the overall infrastructure, it improves separation of concerns as well. That pretty much resembles the paradigm and main goal of role-based programming. Although, the concept of roles has been around for decades, starting in the field of databases [17], the research landscape on it is very diverse and fragmented.

Over time researchers have proposed several implementation approaches targeting the contextual nature of roles and their representation at runtime. Unfortunately, until today there is no common definition of what a role actually is. Most of the resulting languages are reinventing the wheel over and over again, implementing different role features for their specific research area [8]. We argue that evolving objects (core objects with addable or removable extension) are the generalization of role-playing objects as a novel reuse and adaptability unit in dynamic collaborations.

The following main features of roles extracted from the literature [8], [18] can be fulfilled by evolving objects as implemented by *SCROLL* while other coeval approaches will fail to do so (this is explained in more depth in Sec. VI).

- **Roles have properties and behaviors.** An extension adds new functionality or state to its core.
- **Objects may acquire and abandon roles dynamically.** Adding and removing new behavior is the main idea of evolving objects. In particular, a role can be transferred from one object to another.
- **Objects may play the same role (type) several times.** With the grouping of extension in compartments as first-class citizens representing contextualized collaborations, one can easily allow for attaching multiple instances of the same extension type in different contexts to one core object.

As a brief example for applying *SCROLL* for role-based programming, see Fig. 5 and 6. We implement a manual transport (class `ManualTransport`) of a `Person` with a `Car` to a certain `Location` by augmenting these core classes with being a `NormalCar` and a `Driver` respectively. For the autonomous transport (class `AutonomousTransport`), it is the `SmartCar` and `Passenger` with different behavior for driving or using the brakes. `Target` and `Source` roles added to locations are used in the actual transportation by the method `travel()` in `TransportationRole`. That role will augment a specific transportation (either the manual or the autonomous one) and alters its behavior. All important API-calls (like the `+-Operator` or `play`) are explained in Sec. II. Note, that the query-function `one[SomeType]()` (e.g., on line 18 in Fig. 5, right side) allows for querying exactly one instance of the given type that should be contained in the current instance of a `Compartment`.

In summary, with respect to the effects on separation of concerns, both role-playing objects and dynamically evolving objects as generalization provide a handy abstraction of context-dependent dynamic behavior and state.

V. Comparison with manual Implementations and Patterns

The following section demonstrates the advantages of the proposed library approach for pure embedding of evolving objects by comparing it to simple, manually instantiated implementations and design patterns widely used when people try to cope with the required dynamics [19]. For a summary, see Table I.

The most basic solution would be to use one **Single Type** for your core object and all extensions. If they do not differentiate in behavior and you do not plan for future changes, that would be a valid solution without any over-engineering. On the downside, that leads to one single complex type, that may be hard to maintain later on. If extensions introduce many different features, one may think about implementing them as **Separate Types**. That removes coupling and unnecessary tangling of relationships between them. Sadly, it introduces the duplication of features and a loss of integrity with shared state and behavior. Using **Subtypes** for every extension and putting the common things into the supertype for each extension may overcome this issue while being conceptually simple. On the downside the resulting inheritance hierarchy may be hard to adapt with multiple or changing extensions as each of them requires the interface of the supertype to be changed as well.

 ISBN: 978-1-61208-532-6

```
class Person(val name: String)
class Car(val licenseID: String)
class Location(val name: String)

class Transportation() extends Compartment {
  object AutonomousTransport extends Compartment
      ↪ {
    class SmartCar() {
      def drive() {
        info("I am driving autonomously!")
      }
    }
    class Passenger() {
      def brake() {
        info(s"I can't reach the brake. I am
            ↪ ${+this name} and just a
            ↪ passenger!")
      }
    }
  }

  object ManualTransport extends Compartment {
    class NormalCar() {
      def drive() {
        info(s"I am driving with a driver called
            ↪ ${+one[Driver]() name}.")
      }
    }
    class Driver() {
      def brake() {
        info(s"I am ${+this name} and I am hitting
            ↪ the brakes now!")
      }
    }
  }

  class TransportationRole(source: Source,
      ↪ target: Target) {
    def travel() {
      val kindOfTransport = this player match {
        case ManualTransport => "manual"
        case AutonomousTransport => "autonomous"
      }
      info(s"Doing a $kindOfTransport
          ↪ transportation with the car
          ↪ ${one[Car]().licenseID} from
          ↪ ${+source name} to ${+target
          ↪ name}.")
    }
  }

  class Target()
  class Source()
}
```

```
new Transportation {
  val peter = new Person("Peter")
  val harry = new Person("Harry")
  val googleCar = new Car("A-B-C-001")
  val toyota = new Car("A-B-C-002")

  new Location("Munich") play new Source()
  new Location("Berlin") play new Source()
  new Location("Dresden") play new Target()

  harry play new ManualTransport.Driver()
  toyota play new ManualTransport.NormalCar()

  +toyota drive()
  ManualTransport play
    new TransportationRole(
      one[Source]("name" ==# "Berlin"),
      one[Target]()
    ) travel()

  peter play new AutonomousTransport.Passenger()
  googleCar play new
      ↪ AutonomousTransport.SmartCar()

  +googleCar drive()
  AutonomousTransport play
    new TransportationRole(
      one[Source]("name" ==# "Munich"),
      one[Target]()
    ) travel()

  +peter brake()
  +harry brake()
}
```

Figure 5. The SmartCar example (*instance code*, top) and the corresponding *model code* (left).

```
I am driving with a driver called Harry.
Doing a manual transportation with the car A-B-C
    ↪ -002 from Berlin to Dresden.
I am driving autonomously!
Doing a autonomous transportation with the car A-
    ↪ B-C-001 from Munich to Dresden.
I can't reach the brake. I am Peter and just a
    ↪ passenger!
I am Harry and I am hitting the brakes now!
```

Figure 6. Running the example generates the *console output* shown above.

The classification of domain objects inheritance introduces is static. An alternative to that, would be to use the **Role-Object-Pattern** [20]. The core object now has a multi-valued association to its extensions as separate types with a common supertype. This is a very direct implementation without the need of changing some interface when introducing new extensions. It can become complicated when dealing with constraints between those extensions and again, with shared state. Additionally, object-schizophrenia needs to be targeted explicitly which applies to extensions when trying to implement them with the Role-Object-Pattern. One has to deal with method call dispatch, encapsulation and object comparison manually [21]. We continue with **Multiple Inheritance or Traits**. Although these concepts are semantically fine to implement extensions, they will lead to a very static system again with an exponential blowup in the number of required classes for every new context one needs to add. Additionally, parallel object hierarchies may occur where cross-tree constraints are very hard to maintain. **Delegation** on the other hand mimics the inheritance mechanism on object level. This requires (the generation of) a lot of management code and leads to object-schizophrenia, too. Finally, **Delegation-Layers** define layers that group behavior for sets of objects and for sets of classes. Sadly, it implies fixed hierarchies and thus a system design that is too static.

 ISBN: 978-1-61208-532-6

Table I. COMPARISON OF APPROACHES FOR ESTABLISHING DYNAMIC OBJECTS AT RUNTIME (SOLELY BASED ON [19]). ■ INDICATES THAT THERE IS A PROBLEM IN THE GIVEN CATEGORY. PLEASE NOTE, THAT THIS COMPARISON ONLY CONSIDERS APPROACHES THAT DO NOT RELY ON CUSTOM COMPILERS, GENERATORS OR OTHER TOOLING.

	Single complex type	Shared State / Behavior	Scalability	Interface	Change	Constraints	Object Schizophrenia	Exponential blowup	Static Design	Parallel Hierarchies	Management Code
Single Type	■		■	■	■				■		
Separate Type		■	■			■					■
Subtype With Internal Flag	■	■	■	■	■			■	■		■
Subtype With Hidden Delegation	■	■	■	■	■			■	■		■
Subtype With State Object		■	■	■	■			■	■		■
Role Object Pattern		■				■	■				■
Multiple Inheritance / Traits			■		■			■	■	■	
Delegation / Delegation Layers		■			■		■				■

VI. RELATED WORK

This section summarizes and compares how different runtime environments or technical spaces could be used to realize evolving objects.

A. Evolving objects with other statically-typed, object-oriented languages

First, *SCROLL* requires the concept of a marker trait, i.e., a mixin to an object for triggering compiler rewrites handing over calls to the library for finding behavior that is not natively available at the core object (trait `Dynamic`, as explained in Sec. III-B). *Second*, a technical solution for assembling a compound object from the core plus all its extension bypassing object-schizophrenia [2] is needed. *Third* requirement is the storage of the relationships between each individual core object and its extensions, which should be easy with any graph-based backend, or alternatively with simple tables or maps. If one is able to find or emulate these three techniques in the desired statically-typed, object-oriented language it is easy to provide an alternative implementation of *SCROLL*. In conclusion, this proposed *implementation pattern* (requiring the before-mentioned three basic technologies) is applicable to many host languages. With the *ExpandoObject* [22] C# can be considered as the most promising option to provide such an implementation of *SCROLL* in another language. The ExpandoObject represents an object that allows for dynamically adding and removing its members at runtime. However, this works at another level of granularity compared to *SCROLL*. Only single members, like a function or an attribute, can be attached or removed at a single point in time. With *SCROLL* you are allowed to group them together (e.g., into classes, case classes or objects) and add or remove all contained members at once. Better separation of concerns is achieved that way.

B. Evolving objects with Aspect- / Subject-oriented programming languages

Aspect-oriented programming allows to implement cross-cutting concerns via join-points and pointcuts. Often the composition is done statically although there exist a few dynamic approaches. **ObjectTeams/Java** (OT/J) [23] uses dynamic aspect weaving at bytecode-level for adding behavior. Subject-oriented programming utilizes different class hierarchies from different perspectives. On the downside there is no real composition language and the set of composition operators is fixed. Furthermore, no real control flow on the composition itself exists.

C. Evolving objects with role-based programming languages

Although dynamically evolving objects are the more general concept compared to role-playing objects as outlined in Sec. IV, we consider them to be useful for a technical realization as well. Interestingly, most of the existing role-based programming languages are extensions to Java. They are either compiled to Java source code [24], [25], [26], [27] or to bytecode [23] directly.

Chameleon [24] provides roles through *constituent methods* allowing to overwrite methods of their players, which work like advices in aspect-oriented programming. As a major drawback of Chameleon its roles extend the player to gain access to it, which is conceptually wrong [18] and limits the flexibility of roles. **Rava** [25] overcomes this by employing the *Role-Object-Pattern* [20] extended with the *Mediator-Pattern* [28]. They use special keywords to steer the generation of management code. Due to the use of the Role-Object Pattern and generation to plain Java, this solution suffers from object-schizophrenia [29]. **JavaStage** [27] solves this problem, by only supporting static roles. They are directly compiled into the players as inner classes. To avoid name clashes, a customizable method renaming strategy is applied. Its main advantages are the capability to specify a list of required methods instead of a specific player class. This approach limits itself to static roles as well, unable to represent their relational and context-dependent nature. **Rumer** [30] offers first-class relationships and modular verification over distributed state. Furthermore, it provides several intra-relationship constraints usable to restrict these relationships. Roles are the named places of a relationship with attributes and methods but without inheritance. Roles are only accessible within a relationship

and not from their player. **ScalaRoles** [31] is probably the closest relative to *SCROLL*. It is implemented as Scala library as well and utilizes dynamic proxies (from the Java API) to implement roles. The practical implementation using Scala's traits as roles reveals the problem that the order of role binding influences the resulting type, e.g., a person playing the father role first and then the student role is another type than the same person playing those roles the other way around. The most sophisticated and mature approach so far is **ObjectTeams/Java** (OT/J) [23]. Like Chameleon above, OT/J allows to override methods of a player by aspect weaving. It introduces *Teams* to represent compartments whose inner classes automatically become roles. Notably, OT/J supports both the inheritance of roles and teams whereas the latter leads to family polymorphism [32]. On the downside, it does neither support multiple unrelated player types for a role type nor first class relationships and only a limited form of constraints. This is similar to **powerJava** [33], which also introduces compartments, denoted *Institutions*, whose inner classes represent roles. However, powerJava features the distinction between role interface and role implementation where the former is callable from outside a specific institution and the latter is the institution-specific implementation of the same interface. Both Rava and powerJava are the only research prototypes providing a working compiler. Sadly, the project has been abandoned [34]. A more recent approach towards context-oriented programming is **NextEJ** [35] as the successor of EpsilonJ [36]. It provides *Contexts* as first class citizens which do not only group roles but also represent an activation scope at runtime. These *context activation scopes* can be nested and act as a barrier where all roles are instantiated and bound automatically. So far, they only published their type-system of the core calculus and no compiler.

In summary, it is necessary to investigate how well the implementation with SCROLL for binding roles as technical realization for evolving objects blends into contemporary approaches. We use an already published classification scheme from the literature [8], [18]. A compact overview is given in Table II. Most of the role features in question are supported.

Table II. COMPARISON OF COEVAL APPROACHES FOR ROLES AT RUNTIME BASED ON 26 CLASSIFYING FEATURES EXTRACTED FROM THE LITERATURE [8], [18]. IT DIFFERENTIATES BETWEEN FULLY (■), PARTLY (⊞) AND NOT SUPPORTED (□) FEATURES.

Feature [8]	**Chameleon** [24]	**OT/J** [23]	**Rava** [25]	**powerJava** [26]	**Rumer** [30]	**ScalaRoles** [37]	**NextEJ** [35]	**JavaStage** [27]	**SCROLL**
1.	■	■	■	■	■	■	■	■	■
2.	□	⊞	□	⊞	■	□	⊞	□	□
3.	■	■	■	■	■	■	■	■	■
4.	■	■	□	■	■	■	■	□	■
5.	■	■	■	⊞	■	■	■	■	■
6.	□	■	□	■	■	□	□	■	■
7.	■	□	■	■	⊞	■	■	■	■
8.	□	■	□	■	□	■	■	■	■
9.	■	□	□	■	□	■	■	□	■
10.	■	■	■	■	■	■	■	■	■
11.	■	■	■	■	■	■	■	■	■
12.	■	■	■	■	■	■	■	■	■
13.	□	■	■	■	□	■	□	■	■
14.	⊞	⊞	□	□	■	■	⊞	□	■
15.	■	■	■	■	□	■	■	■	■
16.	□	□	□	□	■	□	□	□	□
17.	□	□	□	□	□	□	□	□	□
18.	□	■	□	□	⊞	⊞	⊞	□	■
19.	□	■	□	⊞	⊞	□	■	□	□
20.	□	■	□	■	■	■	■	□	■
21.	□	□	□	■	□	⊞	■	□	■
22.	□	■	□	□	■	□	□	□	■
23.	□	■	□	□	□	□	□	□	■
24.	□	■	□	⊞	■	■	■	□	■
25.	□	■	□	□	□	■	□	□	■
26.	□	■	□	⊞	■	■	■	□	■

VII. FUTURE WORK

Several developments are currently work in progress or targeted for investigation in the near future. Because the actual set of dynamic extensions that are bound to a core object can not be statically determined, static type-safety is lost at a certain point as already mentioned in Sec. III-B. The *SCROLLCompilerPlugin* [38] is a plugin for the standard Scala compiler and runs right after its typing phase. It allows for validating the source code (i.e., traversing the syntax tree) and generates meaningful warnings and errors, e.g., if the developer is requesting behavior from a dynamic extension that was never bound. In interdisciplinary collaborations, we aim for other use-cases for applying the concept of dynamically evolving objects. They should help the domain expert to cope with its specific implementation concerns. Specifically in systems biology, and more generally in scientific computing (e.g., with a Next-Generation Parallel Particle-Mesh Language [39]) using this concept looks promising. The separation of concerns achieved this way greatly improves the quality of code written in these field of research. With respect to the required performance, methods for translating the specific binding and behavior-lookup for dynamic objects to a native and fast performing technological platform need to be developed. Another promising direction is the investigation of the `invokedynamic` bytecode keyword introduced with Java 7 to provide an alternative to *SCROLL*. An appropriate implementation and comparison of those two approaches in terms of runtime-efficiency and improvement design-time development experience is currently targeted.

VIII. CONCLUSIONS

In summary, this work presents an attempt to bridge the gap between statically-typed, object-oriented languages and evolving objects at runtime by introducing *SCROLL* as a lightweight library that allows for pure embedding of dynamically evolving objects in a modern, statically typed object-oriented language. Arbitrary objects can be augmented with extensions allowing for adding and removing behavior and state at runtime. They are combined to one logical compound object through the library solving object-schizophrenia. The library allows for easy integration of existing (Java Virtual Machine based) legacy code and a high separation of concerns, e.g., when applied to roles in contexts. Ultimately, following the rules of the proposed implementation pattern as the core idea of *SCROLL* one could easily implement a very similar library in another host language.

 ISBN: 978-1-61208-532-6

ACKNOWLEDGMENT

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing". Special thanks go to Sebastian Götz, Ulrike Schöbel and Anthony Sloane for improving this paper.

REFERENCES

[1] P. H. Menon, Z. Palmer, A. Rozenshteyn, and S. Smith, "Types for flexible objects," Technical report, The Johns Hopkins University, Tech. Rep., 2013.

[2] U. Aßmann, Invasive Software Composition. Springer-Verlag, 2003.

[3] J. F. Furrer, "Zukunftsfähige Softwaresysteme," Informatik-Spektrum, 2015, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1007/s00287-015-0909-6

[4] M. Leuthäuser, "SCROLL," https://github.com/max-leuthaeuser/scroll, 2016, [last viewed 01.12.2016, 09.00].

[5] EPFL, "Scala Website," http://www.scala-lang.org/, 2016, [last viewed 01.12.2016, 09.00].

[6] E. Meijer and A. Peter Drayton, "Static typing where possible," Dynamic Typing When Needed: The End of the Cold War Between Programming Languages, 2004.

[7] M. Odersky, L. Spoon, and B. Venners, "Programming in scala: a comprehensive stepby-step guide," Artima Inc, August, 2008.

[8] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann, "A metamodel family for role-based modeling and programming languages," in Software Language Engineering, ser. Lecture Notes in Computer Science, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds. Springer International Publishing, 2014, vol. 8706, pp. 141–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11245-9_8

[9] M. Leuthäuser, "SCROLL Examples," https://github.com/max-leuthaeuser/SCROLL/tree/master/examples/src/main/scala/scroll/examples, 2016, [last viewed 01.12.2016, 09.00].

[10] B. Harrison, "Subject-oriented Programming vs. Design Patterns," http://www.research.ibm.com/sop, 1997, [archived as of May 1997].

[11] C. Dony, J. Malenfant, and P. Cointe, "Prototype-based languages: From a new taxonomy to constructive proposals and their validation," in Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 201–217. [Online]. Available: http://doi.acm.org/10.1145/141936.141954

[12] EPFL, "Scala Dynamic Trait SIP," http://docs.scala-lang.org/sips/completed/type-dynamic.html, 2016, [last viewed 01.12.2016, 09.00].

[13] EPFL, "Scala Dynamic Trait ScalaDoc," https://github.com/scala/scala/blob/2.12.x/src/library/scala/Dynamic.scala, 2016, [last viewed 01.12.2016, 09.00].

[14] B. Naveh and Contributors, "jGraphT," http://jgrapht.org/, 2016, [last viewed 01.12.2016, 09.00].

[15] Google, "Guava," https://github.com/google/guava, 2016, [last viewed 01.12.2016, 09.00].

[16] A. M. Sloane, L. C. Kats, and E. Visser, "A pure embedding of attribute grammars," Science of Computer Programming, vol. 78, no. 10, 2013, pp. 1752–1769.

[17] C. W. Bachman, "The programmer as navigator," Commun. ACM, vol. 16, no. 11, 1973, pp. 635–658.

[18] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," Data & Knowledge Engineering, vol. 35, no. 1, 2000, pp. 83–106.

[19] M. Fowler, "Dealing with roles," in Proceedings of PLoP, vol. 97, 1997.

[20] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, "The role object pattern," in Washington University Dept. of Computer Science, 1997.

[21] S. Herrmann, "Demystifying object schizophrenia," in Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance. ACM, 2010, p. 2.

[22] Microsoft, "Expando Object," https://msdn.microsoft.com/en-us/magazine/ff796227.aspx, 2016, [last viewed 01.12.2016, 09.00].

[23] S. Herrmann, "Programming with roles in ObjectTeams/Java." AAAI Fall Symposium, Tech. Rep., 2005.

[24] K. B. Graversen and K. Østerbye, "Implementation of a role language for object-specific dynamic separation of concerns," in AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies, 2003.

[25] C. He, Z. Nie, B. Li, L. Cao, and K. He, "Rava: Designing a java extension with dynamic object roles," in Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on. IEEE, 2006, pp. 7–pp.

[26] M. Baldoni, G. Boella, and L. van der Torre, "Roles as a coordination construct: Introducing powerjava," Electr. Notes Theor. Comput. Sci, vol. 150, no. 1, 2006, pp. 9–29.

[27] F. S. Barbosa and A. Aguiar, "Modeling and programming with roles: introducing javastage," Instituto Politécnico de Castelo Branco, Tech. Rep., 2012.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[29] S. Herrmann, "Demystifying object schizophrenia," in Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance, ser. MASPEGHI '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:5.

[30] S. Balzer, T. Gross, and P. Eugster, "A relational model of object collaborations and its use in reasoning about relationships," in ECOOP, ser. Lecture Notes in Computer Science, E. Ernst, Ed., vol. 4609. Springer, 2007, pp. 323–346.

[31] M. Pradel and M. Odersky, "Scala roles: Reusable object collaborations in a library," in Software and Data Technologies. Springer Berlin Heidelberg, 2009, pp. 23–36.

[32] S. Herrmann, C. Hundt, and K. Mehner, "Translation polymorphism in object teams," TU Berlin, Tech. Rep., 2004.

[33] E. Arnaudo, M. Baldoni, G. Boella, V. Genovese, and R. Grenna, "An implementation of roles as affordances: powerJava," Aug. 31 2009.

[34] G. Wielenga, "On powerjava: "roles" instead of "objects"," https://blogs.oracle.com/geertjan/entry/on_powerjava_roles_instead_of, jan 2013, [Online; accessed 28-May-2014].

[35] T. Kamina and T. Tamai, "Towards safe and flexible object adaptation," in International Workshop on Context-Oriented Programming. ACM, 2009, p. 4.

[36] T. T. S. Monpratarnchai, "The design and implementation of a role model based language, EpsilonJ." in Proceedings of the 5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2008), 2008.

[37] M. Pradel and M. Odersky, "Scala Roles - A lightweight approach towards reusable collaborations," in International Conference on Software and Data Technologies (ICSOFT '08), 2008.

[38] M. Leuthäuser, "SCROLLCompilerPlugin," https://github.com/max-leuthaeuser/SCROLLCompilerPlugin, 2016, [last viewed 01.12.2016, 09.00].

[39] S. Karol, P. Incardona, Y. Afshar, I. F. Sbalzarini, and J. Castrillon, "Towards a next-generation parallel particle-mesh," in Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015), T. van der Storm and S. Erdweg, Eds., 2015, vol. abs/1508.03536, pp. 7–8. [Online]. Available: http://arxiv.org/abs/1508.03536

 ISBN: 978-1-61208-532-6